Leveraging Transactional Execution for Memory Consistency Model Emulation

Ragavendra Natarajan, University of Minnesota
Antonia Zhai, University of Minnesota

System emulation is widely used in today's computer systems. This technology opens new opportunities for resource sharing as well as enhancing system security and reliability. System emulation across different instruction set architectures (ISA) can enable further opportunities. For example, cross-ISA emulation can enable workload consolidation over a wide range of microprocessors and potentially facilitate the seamless deployment of new processor architectures. As multicore and manycore processors become pervasive, it is important to address the challenges towards supporting system emulation on these platforms. A key challenge in cross-ISA emulation on multicore systems is ensuring the correctness of emulation when the guest and the host memory consistency models differ. Many existing cross-ISA system emulators are sequential, thus they are able to avoid this problem at the cost of significant performance degradation. Recently proposed parallel emulators are able to address the performance limitation, however, they provide limited support for memory consistency model emulation.

When the host system has a weaker memory consistency model compared to the guest system, the emulator can insert memory fences at appropriate locations in the translated code to enforce the guest memory ordering constraints. These memory fences can significantly degrade the performance of the translated code. Transactional execution support available on certain recent microprocessors provides an alternative approach. Transactional execution of the translated code enforces sequential consistency (SC) at the coarse-grained transaction level, which in turn ensures that all memory accesses made on the host machine conform to SC. Enforcing SC on the host machine guarantees that the emulated execution will be correct for any guest memory model. In this paper, we compare and evaluate the overheads associated with using transactions and fences for memory consistency model emulation on the Intel Haswell processor. Our experience of implementing these two approaches on a state-of-the-art parallel emulator, COREMU, demonstrates that memory consistency model emulation using transactions performs better when the transaction sizes are large enough to amortize the transaction overhead, and the transaction conflict rate is low; while inserting memory fences is better for applications in which the transaction overhead is high. A hybrid implementation that dynamically determines which approach to invoke can outperform both approaches. Our results, based on the SPLASH-2 and the PARSEC benchmark suites, demonstrate that the proposed hybrid approach is able to outperform the fence insertion mechanism by 4.9%, and the transactional execution approach by 24.9% for 2-thread applications; and outperform them by 4.5% and 44.7%, respectively for 4-threaded execution.

Categories and Subject Descriptors: D.3.4 [**Programming Languages**]: Processors – Code Generation, Run-time Environments; D.1.3 [**Software**]: Concurrent Programming – Parallel Programming; C.1.4 [**Processors Architectures**]: Parallel Architectures

Additional Key Words and Phrases: Parallel Emulators, Memory Consistency Models, Transactional Memory

ACM Reference Format:
Ragavendra Natarajan, and Antonia Zhai, 2015. Leveraging Transactional Execution for Memory Consis-

---

This article presents new work and is not an extension of a conference paper.
Author's addresses: R. Natarajan, and A.Zhai, Department of Computer Science and Engineering, University of Minnesota; emails: {natar, zhai}@cs.umn.edu

DOI: http://dx.doi.org/10.1145/0000000.0000000

tency Model Emulation. *ACM Trans. Architec. Code Optim.* V, N, Article XXXX (May 2015), 25 pages.
DOI: http://dx.doi.org/10.1145/2786980

1. INTRODUCTION

System virtualization, or system emulation, is a key technology that is widely used in today's computers. Data centers reduce costs by employing virtualization in order to utilize computational resources more efficiently. Virtualization also provides strong isolation between different applications running on the same hardware, thereby resulting in better security and reliability in the cloud. System emulation has numerous applications beyond cloud computing as well. Emulation is widely used as a safe way to examine malware. Emulation also facilitates execution migration of applications across different platforms and devices. Support for emulation across processors with different instruction set architectures (ISA) can open up further opportunities in many different applications of system emulation.

Cross-ISA emulation can help data centers to consolidate workloads over a wider range of processors. It can also enable new processor architectures to be deployed easily in data centers without any changes to existing applications. For example, x86-based applications can take advantage of servers built with emerging low-power processors with different ISAs. Cross-ISA emulation support in data centers in turn can potentially open numerous avenues. It can enable data centers to offer Platform-as-a-service (PaaS) on multiple processor architectures irrespective of the underlying server hardware. It can enable wider adoption of ubiquitous computing, which harnesses the cloud to run mobile applications, by supporting virtual execution of mobile applications on cloud servers with different ISAs.

Cross-ISA emulation also has potential applications beyond the data center. It can facilitate the execution of incompatible applications on desktop and mobile phones, as well as allow application execution to migrate between different devices seamlessly. For example, it can allow applications developed for ARM mobile processors to run on x86 mobile processors (and vice-versa). Cross-ISA emulation is a crucial development and debugging tool for software developers aiming to port applications to multiple devices with different ISAs. Architects of new ISAs can leverage cross-ISA emulation to emulate the new architecture on existing machines. Cross-ISA system emulation can also allow legacy applications written for older ISAs to run on current systems.

Recent advances in semiconductor technology have resulted in multicore and heterogeneous multicore processors that drive systems from servers to mobile phones. In response to this trend developers are exploiting parallelism in applications. With parallel applications becoming more ubiquitous, cross-ISA virtualization of multithreaded programs is crucial. Although a large body of research exists on system virtualization, relatively few of the prior works address the challenges unique to multithreaded applications. One of the key challenges of virtualizing multithreaded applications across ISAs is ensuring that a program written for the guest system is executed correctly on the host system when the memory consistency models of the two ISAs differ.

The memory consistency model of a processor defines how the results of memory accesses in a program will appear to the programmer. The most intuitive memory consistency model is the *sequential consistency* (SC) model [Lamport 1979] which specifies that the memory operations from a processor appear to execute atomically and in the order they are specified in the program. Enforcing sequential consistency, however, prohibits a number of architecture optimizations crucial to high performance. Therefore, most modern processors choose to implement *relaxed* memory consistency models which are weaker than SC. However, they provide special memory fence instructions as a means to enforce SC. Table I shows the ordering constraints enforced in some modern processor architectures compiled from previous studies [Adve and Gharachor-

Relaxation	W → R order	W → W order	R → RW order
SC			
x86-TSO	✓		
SPARC-TSO	✓		
SPARC-PSO	✓	✓	
SPARC-RMO	✓	✓	✓
POWER	✓	✓	✓
ARM	✓	✓	✓

Table I: Relaxed memory consistency models of modern processors compared to SC. A ✓indicates the corresponding constraint is relaxed.

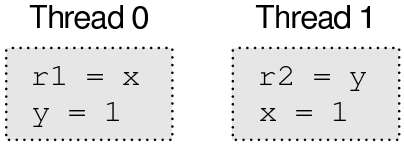


Fig. 1: Pseudocode of an x86 guest application emulated on a POWER host system. All variables have an initial value of 0.

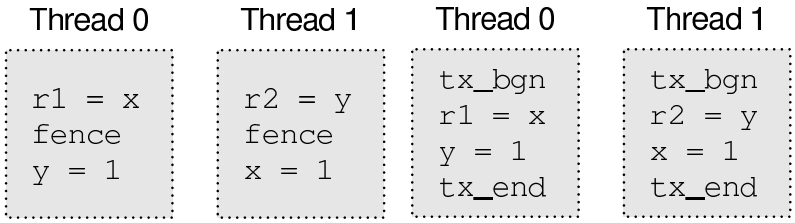
loo 1996; Owens et al. 2009; Maranget et al. 2012]. Different architectures vary in the ordering constraints they relax compared to SC. If the guest and host systems in a virtual environment have different memory consistency models, then it can lead to an incorrect execution of the guest application [Smith and Nair 2005]. An emulated execution is considered incorrect if the order of memory operations that occurred during the actual execution on the host, could *not* have occurred on the guest system.

Consider the pseudocode shown in Figure 1 involving two threads (`Thread 0` and `Thread 1`) and two shared variables (`x` and `y`). `Thread 0` reads the value of `x` into a local variable `r1`, and writes to `y`, while `Thread 1` reads the value of `y` into a local variable `r2`, and writes to `x`. All the variables have an initial value of 0. Assume that the program is executed on an emulated x86 machine running on a POWER host system. Note that the x86 and POWER memory consistency models differ (Table I). Table II shows all the possible values of `r1` and `r2` at the end of the execution of the program. It also indicates the outcomes that are valid under the x86 and the POWER memory consistency models. Under the x86 model, the final outcome of `r1 = 1` and `r2 = 1` is illegal since the outcome requires the stores to `x` and `y` to be reordered before the loads to `r1` and `r2` in both the threads, which is not possible since the x86 model ensures that stores are not reordered with preceding loads (R→W order is not relaxed). However, all the possible outcomes are valid on POWER since the memory consistency model does not guarantee any ordering among the memory accesses. Hence, the virtualized x86 system can observe an illegal result (`r1 = 1` and `r2 = 1`). Therefore, in a cross-ISA virtualized environment, if the guest system has a stronger memory consistency model than the host system, it can lead to an incorrect execution. However, if the guest system has a weaker memory consistency model than the host system (e.g. POWER on x86), then the execution is guaranteed to be correct.

Existing cross-ISA system emulators [Bellard 2005; Magnusson et al. 2002] circumvent this issue by executing multithreaded programs sequentially - by emulating a multicore guest system through time-sharing using a single core on the host system. However, such emulators do not harness the power of multicore processors since they are not parallel. Recently proposed parallel emulators use multiple cores on the host

Result	x86-TSO	POWER
r1 = 0, r2 = 0	✓	✓
r1 = 0, r2 = 1	✓	✓
r1 = 1, r2 = 0	✓	✓
r1 = 1, r2 = 1	×	✓

Table II: Effect of the memory consistency model on the result of Figure 1



(a) using fences (b) using transactions

Fig. 2: Correct x86 emulation on the POWER host system.

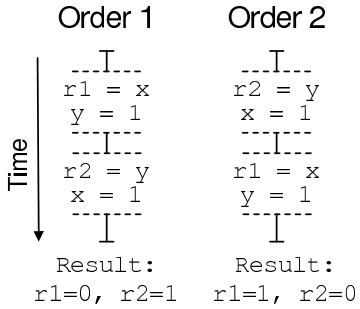


Fig. 3: Possible orderings of memory accesses using transactions on the POWER host system.

system to emulate multicore guest systems [Wang et al. 2011; Ding et al. 2011]. Consequently, they are much faster than sequential emulators. However, they only support emulation of guest and host systems with the same ISA (e.g. x86 on x86), or a guest system with a weaker memory consistency model than the host system (e.g. ARM on x86).

In order to ensure correct emulation when the guest system has a stronger memory consistency model than the host system, an emulator must insert memory fences in the translated host code at runtime. For the example shown in Figure 1, a memory fence must be inserted between the load to `r1` (`r2`) and the store to `y` (`x`) in the translated POWER host code by the emulator. The memory fence ensures that the load and store in a thread do not get reordered. Figure 2(a) shows the pseudocode of the correct translated host code.

Parallel emulators can adopt *transactional execution* as an alternative to runtime fence insertion in order to ensure correct emulation. Consider the pseudocode of the translated POWER host code shown in Figure 2(b) which is identical to the guest application in Figure 1, except now the parallel region is transactionally executed on the host system by each thread. Transactional execution ensures that all memory accesses within a transaction appear to be executed atomically and isolated from other

transactions. It also ensures that transactions from the same thread are executed in order. Therefore, transactional execution effectively enforces sequential consistency on the host system. The only two possible orderings of the guest code on the POWER host system and the corresponding results, both of which are legal on the x86 guest memory consistency model, are shown in Figure 3.

Parallel emulators can leverage either Hardware Transactional Memory (HTM), or Hardware Lock Elision (HLE) that are supported on recent processors [Intel Corporation 2013; Jacobi et al. 2012; Cain et al. 2013] in order to realize transactional execution. HTM allows for speculative execution of code regions, memory conflict detection, and rollback/replay of aborted transactions. HLE allows critical sections to execute speculatively and concurrently [Rajwar and Goodman 2001]. A speculative critical section is aborted and re-executed normally when a conflict is detected by the hardware. Therefore, the parallel region in Figure 2(b) can be executed inside either a transaction (HTM), or a lock-elided critical section (HLE), with the same results. However, there can be implementation differences depending on whether the HTM support automatically ensures forward progress of aborted transactions, or requires the programmer to ensure forward progress by explicitly specifying *fallback* code to be executed on a transaction abort. We clarify the implementation differences in Section 3.2.

There are tradeoffs involved between using memory fences and transactions for memory consistency emulation. Emulation using memory fences allows selective enforcement of certain memory ordering constraints so as to exactly emulate the guest memory consistency model. Emulation using transactions, on the other hand, guarantees that the emulated execution will be correct on any guest memory consistency model by enforcing SC on the host system. Although transactional emulation enforces a stricter constraint than necessary, it can outperform emulation using memory fences under certain conditions. Previous studies have shown that fences are not always necessary at runtime if the accesses to shared data made by the threads in a program do not conflict with each other [von Praun et al. 2006; Lin et al. 2010]. Therefore, when emulating using fences, the translated host code incurs the overhead of a fence instruction, which is high [Duan et al. 2009; Lin et al. 2010], even when it is unnecessary. Unlike fences, which incur a fixed cost on every execution, transactions incur a variable cost depending on the abort rate. If there are no conflicts between the threads operating on shared data, then all the transactions will commit without any aborts, resulting in better performance.

In this paper we focus on the problem of supporting system virtualization of a guest system with a stronger memory consistency model than the host system. To the best of our knowledge, this is the first work to explore the problem of supporting cross-ISA system virtualization of guest and host systems with different memory consistency models. This paper makes the following contributions:

— We discuss the tradeoffs involved in using memory fences and transactions for memory consistency model emulation and characterize the overhead of using memory fences and transactions on a recent processor. Our characterization shows that transactional emulation is a viable alternative to using memory fences for memory consistency model emulation.
— We implement the two approaches on a recently proposed parallel emulator and highlight the implementation issues. We propose a hybrid technique that switches between using fences and transactions depending on the application characteristics in order to minimize the overhead.
— We evaluate the overhead of the two approaches and our hybrid technique on a set of real-world parallel applications. The results show that, on average, emulation using the proposed hybrid technique is 4.9% faster than emulation using fences, and 24.9%

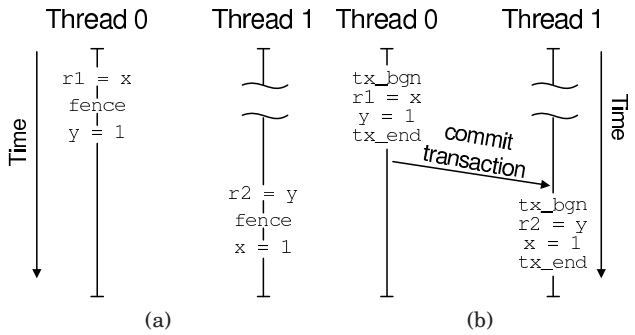


Fig. 4: A correct execution of translated POWER host code (a) without the need for memory fences, (b) without any transaction aborts.

faster than emulation using transactions for 2-thread applications. The corresponding numbers for 4-thread applications are 4.5% and 44.7%, respectively

The rest of the paper is organized as follows. We begin by discussing the tradeoffs involved in using fences and transactions for memory consistency model emulation and characterize the overhead of memory fences and transactions in a recent processor using a set of kernels in Section 2. We then discuss the issues involved in implementing the two techniques in an emulator and propose a hybrid technique which selects the best approach based on the program characteristics in Section 3. We evaluate the performance of the two approaches and our hybrid technique on real world parallel applications in Section 4. We discuss the related work in Section 5 and conclude in Section 6.

2. ELIMINATING CONSISTENCY VIOLATIONS USING MEMORY FENCES AND TRANSACTIONS

In this section we elaborate on how memory fences and transactions can be used to emulate a stronger guest memory consistency model on a host system with a weaker memory consistency model. We discuss the tradeoffs involved in using the two techniques in detail. We characterize the overhead of using memory fence instructions and transactions on a recent processor by using a set of kernels.

2.1. Memory Fences

When emulating a guest system on a host system with a weaker memory consistency model, the emulator must ensure that the guest memory consistency model is faithfully emulated on the host system. One way of ensuring that certain memory orderings are enforced is by inserting fences in the translated host code at runtime. Memory fences are expensive instructions that take tens of cycles to execute on average since they stall the processor until all the memory accesses issued before the memory fence are completed. Finding a correct and efficient placement of memory fences for a program is a challenging task [Burckhardt et al. 2007; Kuperstein et al. 2010; Fang et al. 2003; Duan et al. 2009]. Inserting fences conservatively results in redundant fences and degrades the performance of the program, while using too few fences can cause incorrect emulation.

Even if the number of fences inserted, and their placement, is optimal, previous studies show that a large fraction of the inserted memory fences are in fact unnecessary at runtime [von Praun et al. 2006; Lin et al. 2010]. Figure 4(a) shows an execution of the POWER host code translated using fences from Figure 2(a). Here `Thread 0` com-

pletes its accesses, and its effects are visible to `Thread 1`, before `Thread 1` executes its own accesses. In this execution, the final result is legal on the x86 guest even without any fences inserted in the translated code, since, even if the accesses made by both threads are reordered on the POWER host system it will not lead to a consistency violation.

2.2. Transactional Execution

Transactional execution support implemented in recent processors provides an alternative method of ensuring correct emulation of a guest system on a host system with a weaker memory consistency model without the use of memory fences. HTM or HLE can be used to group the accesses made by the translated host program into coarse-grained transactions. Hardware support ensures that all memory accesses within a transaction appear to execute atomically and in isolation. It also guarantees that all the transactions executed by the same thread are sequentially ordered. Therefore, transactional emulation guarantees sequential consistency at the coarse-grained transaction level. Consequently, all the memory accesses made by the guest application on the host system are also sequentially consistent. Enforcing sequential consistency on the host machine ensures that the emulated execution is guaranteed to be correct on any guest memory consistency model. Note that the granularity of the transactions does not affect correctness although it can impact performance, and that the accesses within a transaction can be reordered while still appearing to conform to sequential consistency.

Although transactional emulation enforces sequential consistency on the host machine it can in fact outperform emulation using memory fences. Unlike fences, which incur a fixed cost on every execution, the cost of a transaction varies depending on the abort rate. If there are no conflicts between the threads during execution, then all the transactions will commit without any aborts. Figure 4(b) shows a conflict-free execution of the POWER host code translated using transactions from Figure 2(b). In this execution, `Thread 0` commits its transaction before `Thread 1` begins executing its own transaction. Therefore, there is no conflict between the transactions and both commit without any aborts. In this execution, the accesses within a transaction can be reordered on the POWER host and the execution would still be correct. The transactional version of the translated code can outperform the fence version since it does not incur the overhead of executing fence instructions.

The transactional emulation can also result in poor performance under certain conditions. Small transactions cannot effectively amortize the overhead of starting and ending a transaction. Thus, they can result in poor performance. However, increasing the transaction size beyond a certain limit leads to diminishing returns. Large transactions can result in conflicts among memory accesses that are well separated in time and that cannot lead to consistency violations in the guest application. Such false conflicts can increase the abort rate of the transactions, thereby resulting in poor performance.

2.3. Overhead Characterization

In this section we characterize the overhead and tradeoffs between using memory fence instructions and transactions on a recent processor. Our test system is a 4-core, 4-thread x86 Haswell processor with HTM and HLE support running at 3.2GHz. We choose the Haswell architecture since it is currently the only x86 processor with transactional execution support. Our evaluation does not characterize the hardware parameters of the transactional execution support implemented in Haswell since this has already been done by previous work [Ritson and Barnes 2013]. We use HLE to implement our transactions (lock elided critical sections). We begin by comparing the overhead of memory fences and transactional execution in the absence of aborts using

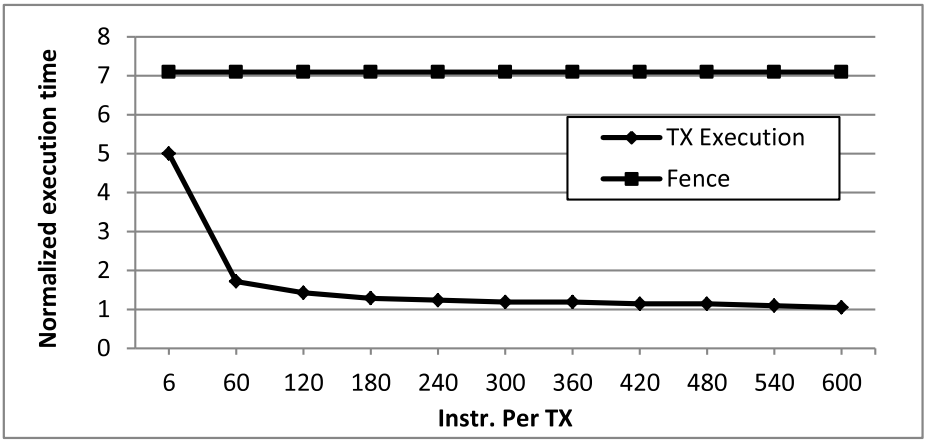


Fig. 5: Execution times of the fence and transactional versions of a sequential micro-benchmark normalized to that of a no-fence and no-transaction baseline across different transaction sizes.

a simple single threaded micro-benchmark. We then evaluate both the correctness and the performance tradeoffs of memory fences and transactional execution using a a set of concurrent, lock-free algorithms.

2.3.1. Overhead: Fences vs. Transactions. We use a single-threaded micro-benchmark to compare the overhead of memory fences and conflict-free transactional execution on Haswell. The micro-benchmark consists of a single loop that iterates 100 million times. Each iteration of the loop performs a store to a memory location, followed by a load from a different memory location. Therefore, the store and load in this micro-benchmark might be executed out of order on x86. We design two versions of the micro-benchmark where this re-ordering is prevented. In the fence version we insert a fence (the x86 `mfence` instruction) between the store and the load, while the transactional version executes each iteration of the loop within a transaction. Note that since the micro-benchmark is single-threaded, there are no aborts due to memory conflicts in the transactional version. Since the loop accesses only a few cache lines, the transactional version does not experience any aborts due to buffer overflows either. We vary the size of each transaction in the transactional version by varying the number of loop iterations executed within each transaction, while keeping the total number of loop iterations constant.

Figure 5 shows the execution time of the fence and the transactional versions of the micro-benchmark normalized to the baseline which does not enforce any ordering. The data is shown for various transaction sizes. Each iteration of the loop contains 6 instructions and we vary the number of iterations within a transaction in steps of 10. The results show that the overhead of memory fences on x86 is considerably high. The overhead of transactional execution, on the other hand, varies depending on the transaction size. When the transaction size is small, the overhead of transactional execution is considerable. However, even at a small transaction size the overhead of using memory fences is much higher. As the transaction size increases, the overhead of transactional execution is amortized and performance improves. Once a large enough transaction size is reached, the overhead of transactional execution is negligible and the performance is comparable to sequential execution. Increasing the transaction size beyond this optimal size does not lead to any performance benefit. These results demonstrate

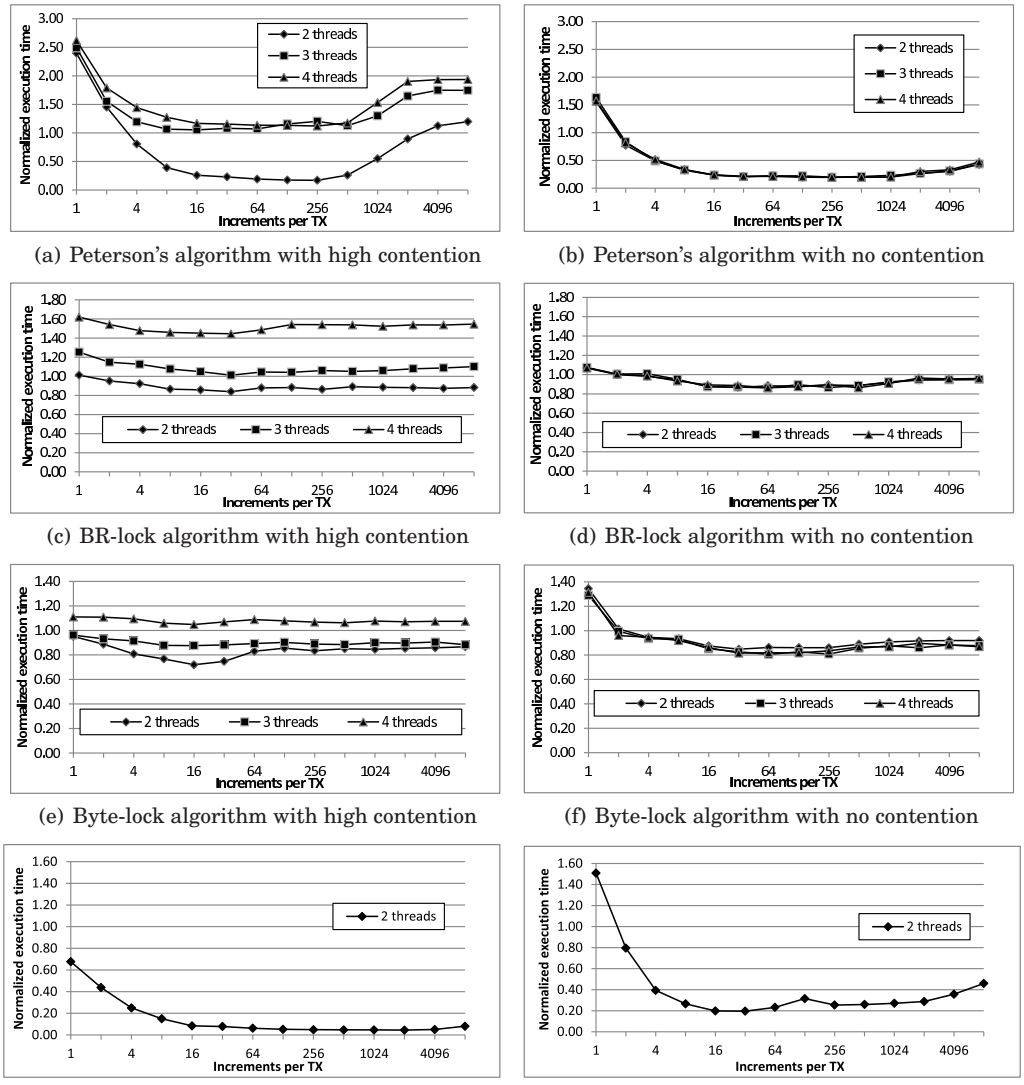
that memory fences are expensive instructions on x86. They also highlight that using transactional execution to enforce memory ordering, instead of memory fences, can lead to substantial performance benefits if the abort rate is low and the transactional overhead is amortized.

2.3.2. Concurrent Micro-Benchmark Results. In order to evaluate both correctness and the performance tradeoffs, we use a set of concurrent, lock-free algorithms which are written assuming SC. Thus, these micro-benchmarks require memory fences for correct execution on x86 machines. All these algorithms enforce mutual exclusion among threads in a multi-threaded program, using only shared memory variables for communication. Each algorithm describes an entry region, which is executed by a thread prior to entering the critical section, and an exit region, which is executed by a thread once it exits the critical section. We briefly describe the kernels below.

— **Peterson's algorithm**: A well known algorithm [Peterson 1981] for enforcing mutual exclusion in a multi-threaded program. The algorithm requires 1 fence in the entry region code for correct execution on the x86 ISA.
— **Big-Reader lock algorithm (BR-lock)**: A reader-writer lock implementation [Bovet and Cesati 2005] originally proposed and used in the Linux kernel. The algorithm requires 2 fences, both in the entry region code, for correct execution on the x86 ISA.
— **Byte-lock algorithm**: Another reader-writer lock implementation proposed in [Dice and Shavit 2010]. The algorithm requires 2 fences, both in the entry region code, for correct execution on the x86 ISA.
— **Dekker's algorithm**: A well known algorithm [Dijkstra 1965] for enforcing mutual exclusion among 2 threads. It requires 2 fences in the entry region code for correct execution on the x86 ISA.

Each kernel is a simple program where multiple threads compete simultaneously to increment a shared variable using a mutex lock implementation listed above. Each thread increments the shared variable a fixed number of times in a loop. One iteration of the main loop involves executing the entry region code, incrementing the shared variable, and executing the exit region code. The threads do not wait between successive increments and therefore, these programs have high contention. We check for correctness by testing the value of the shared variable at the end of program execution to confirm that there were no violations of mutual exclusion. Two versions are implemented for each kernel: a fence version that uses memory fences, and a transactional version (with no fences). In the transactional version of the program, each iteration of the main loop is performed as a single transaction by a thread. We vary the size of a transaction by varying the number of iterations executed within a transaction, while keeping the total number of iterations constant. The number of iterations is varied by unrolling the main loop as many times within each transaction. Note that the entry and exit region codes are executed as many times as the number of increments of the shared variable in each transaction. Although the kernels are not representative of real-world applications, they are useful in order to simulate the conditions under which transactional execution can outperform fences on a real machine.

Effect of transaction size: Figure 6 (a, c, e, g) shows the execution time of the transactional version of each program normalized to the fence version, across different transaction sizes. The data is shown for 2, 3, and 4 threads. Only two thread results are shown for Dekker's algorithm since it cannot be implemented for more than 2 threads. We choose the fence version as the baseline in order to compare the relative performance of memory fences and transactional execution. Since the micro-benchmarks encounter a livelock in the absence of memory fences, we do not choose micro-benchmarks



(a) Peterson's algorithm with high contention

(b) Peterson's algorithm with no contention

(c) BR-lock algorithm with high contention

(d) BR-lock algorithm with no contention

(e) Byte-lock algorithm with high contention

(f) Byte-lock algorithm with no contention

(g) Dekker's algorithm with high contention

(h) Dekker's algorithm with no contention

Fig. 6: Execution time of the kernels using transactions normalized to execution time using memory fences under low and high contention for different transaction sizes.

without fences as the baseline. For all the programs, as the transaction size increases, the performance improves until an optimal size and then begins to drop. Larger transactions amortize the overhead of starting and ending a transaction thereby resulting in better performance. However, very large transactions also increase the possibility of conflicts between the threads, which in turn increases the abort rate of the transactions. A large transaction can also fail if the number of unique cache lines accessed within the transaction exceeds a hardware specific maximum read/write size [Ritson and Barnes 2013]. However, this phenomenon is not observed in the evaluated kernels since each of them accesses just a few unique cache lines within a transaction.

Peterson's, BR-lock, Byte-lock and Dekker's kernels access 3, 3, 2 and 3 unique cache lines within a transaction, respectively. Therefore, the transactions in these kernels abort only due to data conflicts resulting from the increase in the number of instructions (loads/stores) per transaction. Some of the drop in the performance at very large transaction sizes is also due to the aggressive loop unrolling necessary to increase the transaction size. The 2 thread results show that the transactional version, even with a suboptimal transaction size, is faster than the fence version. The execution time of the optimal transactional version of Dekker's, Peterson's, BR-lock, and Byte-lock, with 2 threads, is 0.05, 0.17, 0.88, and 0.83 times the execution time of the fence version, respectively.

Effect of conflict rate: All these programs have a high conflict rate between the threads, and as we increase the number of threads it further increases the possibility of a conflict. A high conflict rate increases the abort rate of the transactions, thereby leading to poor performance. Figure 6 (a, c, e, g), shows that the performance of the transactional version drops significantly compared to the fence version for 3 and 4 threads. The execution time of the optimal transactional version of Peterson's, BR-lock, and Byte-lock, with 4 threads, is 1.12, 1.45, and 1.05 times the execution time of the fence version, respectively. In order to see the performance of transactional execution when there are no conflicts, we modified the kernels (both the fence and the transaction versions) such that each thread operates on a private lock and increments a private variable. Since the transaction support on Haswell tracks dependencies at the cache block level, false sharing among threads can also result in conflicts. Therefore, we take care to place all the private locks and variables on different cache blocks so as to eliminate any false sharing. Figure 6 (b, d, f, h) summarizes the results for all the kernels with 2, 3 and 4 threads. The results show that the performance of the transactional version gets better as the transaction size increases. However, under no contention, there is no drop in the performance of the transactional version at large transaction sizes. The dip in performance observed in the kernels at very large transaction sizes is due to the aggressive loop unrolling required to generate large transactions. Moreover, the performance does not vary with the number of threads when there is no contention. The execution time of the optimal transactional version of Dekker's, Peterson's, BR-lock, and Byte-lock, with 2 threads, is 0.2, 0.2, 0.88, and 0.85 times the execution time of the fence version, respectively. The corresponding numbers with 4 threads for Peterson's, BR-lock, and Byte-lock are 0.2, 0.86, and 0.82, respectively. Even as the number of threads increases, the transactional version is faster than the fence version.

These results show that transactional execution is a viable alternative to using fences in order to emulate a stronger guest memory consistency model on a host with a weaker memory consistency model. If the transaction sizes are large enough to amortize the transaction overhead, and the conflict rate among the threads is low, then transactions can outperform fences. However, if the transactions are too small, or if the program has a high conflict rate, then emulation using memory fences can result in better performance. Therefore, a hybrid technique that can intelligently employ transactions or memory fences for emulation depending on the application characteristics will likely yield the best performance.

3. MEMORY CONSISTENCY MODEL EMULATION

In this section we first briefly describe key techniques in system emulation. We then discuss the implementation issues involved with using fences and transactions for memory consistency model emulation. We then propose a hybrid technique that uses both fences and transactions for emulation in order to minimize the overhead.

System emulators commonly use dynamic binary translation to convert guest assembly instructions to host instructions. The guest code is translated on-the-fly, one basic block at a time. Once a basic block has been translated, it is executed on the host system and the emulator then begins translating the subsequent basic block. Emulators use a *translation cache* to store recently translated *translation blocks*. When translating a guest basic block, the emulator first searches for a corresponding translation block in the translation cache. On a cache miss, the guest block is translated and inserted into the translation cache before execution. Emulators also link translation blocks that are frequently executed in succession, thereby forming *traces*. Traces allow execution to directly jump from one translation block to the next without having to switch from the translation cache to the emulator code in between, thereby speeding up emulation.

3.1. Emulation Using Memory Fences

Automatic insertion of fence instructions in parallel programs to eliminate memory consistency violations is a well known problem. Prior works propose compiler techniques that automatically insert fences, or tools that provide the programmer with information about possible memory consistency violation bugs in the program [Burckhardt et al. 2007; Kuperstein et al. 2010; Fang et al. 2003; Duan et al. 2009]. These techniques rely on static or dynamic program analysis, memory model descriptions or program inputs. Unfortunately, such high level information is inaccessible to a system emulator at translation time. Moreover, these techniques have a high cost in terms of computation time and therefore are not suitable for integration in a system emulator where dynamic binary translation must be fast. During binary translation the emulator does not have access to information that can help decide whether an access to a memory address is to a private or a shared variable. It also does not have information about the semantics of the guest application that is being translated. Therefore, the emulator must be conservative and insert a memory fence after *every* guest application memory operation in order to ensure correctness [Smith and Nair 2005]. Depending on the number of memory operations in an application, this can lead to a considerable slowdown.

Fences must be selectively inserted only to bridge the gap between the guest and the host memory consistency models. Therefore, certain optimizations can be used to reduce the number of memory fences inserted depending on the guest and the host system. For example, if the guest system is an x86 machine emulated on a POWER host system, then the emulator needs to enforce only R$\rightarrow$R, R$\rightarrow$W and W$\rightarrow$W order on the host system (Table I). Therefore, the emulator must insert a fence after every read operation. A fence is required only *between* two write operations. While inserting fences only after a specific type (read/write) of memory access can be easily implemented in an emulator, inserting fences only *between* specific types of memory accesses is harder. For example, it might not be possible to insert a fence between the last write in a translation block and the first write in the successive block. This is because there might be multiple translation blocks that could potentially be executed after a given translation block. Therefore, the last write in a translation block can be followed by a read or a write in a successive block. Moreover, translation blocks that are executed successively might be translated at different times depending on when they are inserted into the translation cache and hence, it might not be possible to infer the first memory operation in a successive translation block at translation time. Therefore, in order to guarantee correctness the emulator must conservatively insert a fence at the end of a translation block if the last memory access is a write, thus negating most of the performance gain due to the optimization. In practice, we find that using simple opti-

mizations such as inserting a fence only after a specific type of memory operation, is just as effective.

3.2. Emulation Using Transactions

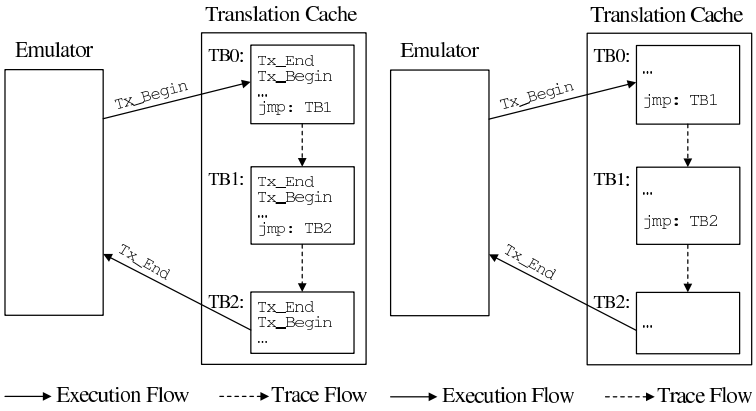
An emulator can also use transactions for memory consistency model emulation. The guest code can be partitioned into chunks and executed as transactions on the host system. The hardware will detect any conflicts among the transactions that are executed simultaneously and re-execute them. Since the emulator cannot be certain if a memory access is to a private or a shared variable, it must enclose *every* memory access in the guest application within a transaction. Therefore, emulation using transactions is equally as conservative as emulation using fence instructions.

The simplest way to form transactions is at the translation block level. However, translation blocks are typically very small and contain only a few instructions. Therefore, executing each translation block as a separate transaction can incur a significant overhead. Executing entire traces as transactions can greatly reduce this overhead since traces typically contain tens of instructions. However, the transaction length is limited by the trace length, which can vary depending on the application.

Figure 7(a) illustrates how the guest code can be partitioned into transactions at the translation block boundaries. The emulator inserts *Tx_begin* and *Tx_end* instructions around each translation block at translation time. If the emulator uses HLE to implement the transactions then it must insert lock-elided lock and unlock instructions instead. A simple approach is to begin every translation block with a {*Tx_end, Tx_begin*} prologue that ends the previous block's transaction and begins the next one. *Tx_begin* and *Tx_end* instructions must be inserted when execution jumps to, and from, the translation cache in order to form complete transactions. Note that transactional execution ensures that there is an implicit fence between the translation blocks.

Forming transactions at the trace level involves a very small change. The emulator inserts *Tx_begin* and *Tx_end* instructions only when execution jumps to, and from, the translation cache but not around every translation block, as shown in Figure 7(b). Transactions must be started at every entry point, and terminated at every exit point, to the translation cache. Although it might be beneficial to form transactions that are larger than the trace size it is not be possible to do this in an emulator environment. All the instructions executed within a trace correspond to the translated guest application. However, when the execution jumps out of the translation cache at the end of a trace, the executed instructions correspond to the emulator code itself. Since only the translated guest code must be executed inside a transaction, a transaction *must* be started and terminated at the beginning and end of a trace, respectively. Thus, any optimization that increases the trace length of an application will also increase the size of the transactions formed.

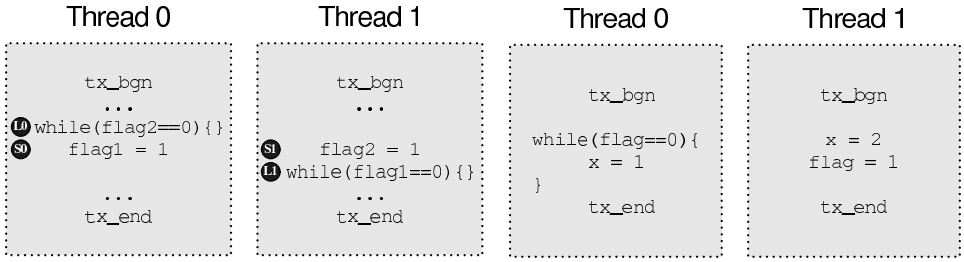
The emulator must generate code differently depending on the hardware support used to implement the transactions. If the transactions are implemented using HLE, then the emulator must start and end each transaction with lock-elided lock and unlock instructions. HLE, which is currently available only on Intel Haswell processors, automatically ensures forward progress on an abort by re-executing the transactions as regular critical sections guarded by atomic locks [Intel Corporation 2013]. Note that the emulator must use the same global lock to guard all the critical sections generated in the code. If HTM is used, then each transaction must start and end with the hardware specific *Tx_begin* and *Tx_end* instructions. Some HTM implementations, such as IBM z/Architecture, provide support to automatically ensure forward progress of aborted transactions [Jacobi et al. 2012]. However, other implementations, such as Intel Haswell and IBM POWER, require the programmer to ensure forward progress by explicitly specifying *fallback* code which is executed on a transaction abort [Intel



(a) Forming transactions at the translation block level

(b) Forming transactions at the trace level

Fig. 7: Forming transactions at the translation block and trace level in an emulator.



(a) Guest application with conditional synchronization emulated using transactions at the trace level. `flag1` and `flag2` are set to 0 initially.

(b) Guest application with conditional synchronization emulated using transactions at the translation block level. `flag` and `x` are set to 0 initially.

Fig. 8: Forward progress issues in transactional emulation.

Corporation 2013; Cain et al. 2013]. In such cases the emulator must generate the fallback code at run-time. The fallback code can point to the original transaction, however this might lead to the program not making any forward progress. Therefore, the emulator must be able to identify when there is no forward progress being made by the program (based on a timeout period or by monitoring the transaction abort rate), and re-translate the code using fences.

Forward progress issues can also arise if the guest application has conditional synchronization. Figure 8(a) shows a guest application with conditional synchronization that has been translated using transactions at the trace level. The variables `flag1` and `flag2` are set to 0 initially. The transactions span multiple basic blocks as shown in the figure. Note that the original guest code might contain fence instructions for the example shown in Figure 8(a), however, the emulator eliminates all fence instructions during translation since the code is emulated using transactions. For correct execution of the program, statement `S1` must complete before loop `L0`, and statement `S0` before loop `L1`. The introduction of transactions, however, requires that `L0` and `S0` execute atomically before `S1` and `L1`, or vice versa. Since the transactions shown in Figure 8(a) are not serializable, this either leads to a live lock or a dead lock. Live locks are possible

even when the translated code has transactions at the translation block boundaries. Figure 8(b) shows an example guest application that has been translated with transactions formed at the translation block boundaries. Both transactions in Figure 8(b) span a single basic block as shown. In this example, it is possible that the store to `x` by `Thread 0` continuously aborts the transaction in `Thread 1`, thereby leading to a live lock. Such forward progress issues are not unique to transactional emulation, and are possible with any application that contains ill-formed transactions as demonstrated by previous studies [Blundell et al. 2005]. The emulator must handle such cases by retranslating the code using fences as described previously. Prior papers which employ transactional execution in a binary translation environment propose a similar solution for detecting when a program is not making any forward progress [Chung et al. 2008].

The guest application may contain user-defined transactions and critical sections. Transaction support implemented in recent processors automatically handles nested transactions by subsuming the inner transaction. One of the advantages of using transactions is that the same approach can work on any host system, as long as it supports transactional execution, since it does not rely on fences. This makes it attractive for emulation where the guest-host configurations can vary.

3.3. Hybrid Emulation Using Memory Fences and Transactions

As characterized in Section 2.3, the overhead of using memory fences and transactions depends on the number of fences inserted at runtime, the conflict rate among threads in the application being emulated, and the size of the transactions formed at runtime. Therefore, a hybrid technique that uses both fences and transactions, and automatically chooses the best approach based on these factors, is likely to provide the best performance.

Such a hybrid technique must estimate the overhead of emulation using transactions and memory fences at runtime. We propose using hardware performance counters to measure the execution time of the translated host code in order to compare the overhead of the two techniques. By measuring the number of host cycles elapsed, the execution time of both versions of the translated host code can be measured accurately. The emulator profiles the overhead of using fences and transactions periodically, and then applies the best policy for emulating the application. Both the policies are profiled for a fixed number of trace executions. During the profiling phase of the fence policy the emulator measures the execution time of the host code translated using memory fences. Once the overhead of fence emulation has been measured, the overhead of transactional emulation is measured similarly. The emulator then makes its decision and applies the best policy for emulation until the next profiling phase.

The main overhead of dynamic profiling is due to translation cache invalidations. Before beginning a profiling phase, the emulator has to invalidate previously translated code and begin translation using the technique being profiled. The translation cache must be invalidated again when the policy being profiled changes. Similarly, once both fence and transaction profiling phases have been completed, the translation cache must be invalidated in order to translate the guest code using the best technique (this can be avoided if the best policy is the same as the policy that is profiled last). However, the overhead of translation cache invalidations is small since each translation block must be translated just once before it is inserted into the translation cache again. The overhead of measuring execution time using hardware performance counters is also negligible. Therefore, the overhead of the dynamic profiling technique is low. The proposed hybrid scheme is simplistic and switches between fence and transactional emulation at a coarse-grained level. A fine-grained hybrid technique that switches between fence and transactional emulation at a per-trace or per-translation block level might yield better performance. However, comparing the execution times of fence and

transactional emulation at a fine-grained granularity also requires fine-grained book-keeping operations. The lack of light-weight hardware performance counters makes the overhead of fine-grained book-keeping operations prohibitively large. The design of an alternate light-weight fine-grained hybrid technique is challenging. A comprehensive treatment of this subject is beyond the scope of this paper.

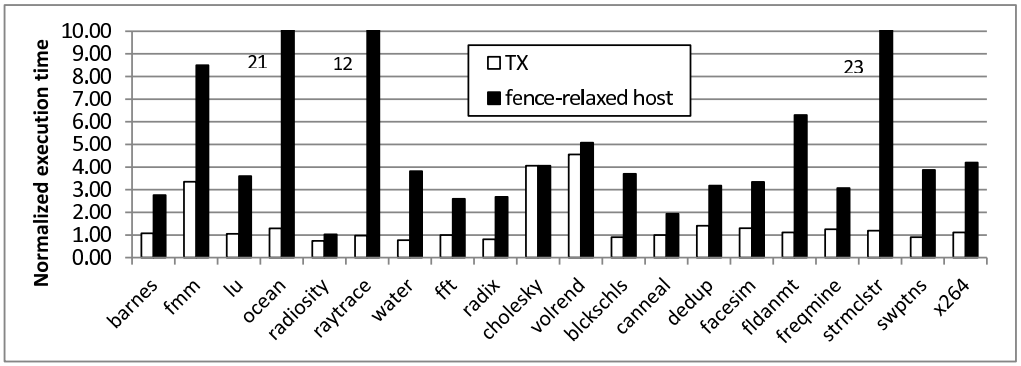
4. EVALUATION

We use COREMU [Wang et al. 2011], a recently proposed parallel emulator for our study. Since COREMU supports only x86 hosts, we use a 4-core, 4-thread, Haswell architecture based, x86 Xeon E3-1225 v3 processor with transaction support as our host system. No modern processor implements a memory model stronger than the x86 memory model. Therefore, in order to simulate a guest system with a stronger memory model we assume a hypothetical sequential consistency guest system with the x86 ISA. We form guest applications for the sequential consistency guest system by taking existing x86 programs and removing all the fence instructions from them. We verify that the sequential consistency guest applications produce incorrect results when emulated on the x86 host system using the unmodified COREMU emulator. We discuss our results in the context of a real cross-ISA system in Section 4.4.

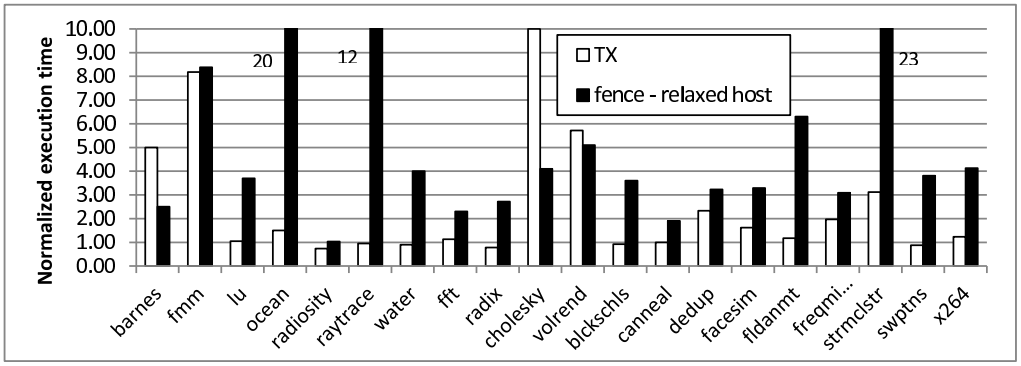
We use two sets of multithreaded guest applications to check for correctness and performance, respectively. The kernels described in Section 2.3 are used to verify correctness. We use eleven (entire set) applications from SPLASH-2 [Woo et al. 1995] and nine applications from PARSEC [Bienia et al. 2008] in order to evaluate the performance overhead of the two techniques. We use the updated input sets from SPLASH-2x and PARSEC-3.0 for our evaluation. Since RAYTRACE is common to both SPLASH-2 and PARSEC we include it just once. We omit BODYTRACK, FERRET and VIPS from PARSEC due to difficulties encountered when running them on COREMU.

We modified COREMU to enforce sequential consistency (the guest memory model) on the host by automatically inserting memory fences after every store instruction in the guest application. Although a memory fence is required only between a store and a load in order to guarantee sequential consistency on an x86 system (to enforce the W→R constraint), such an optimization does not benefit much (Section 3.1). In practice, we find that inserting a fence after every store is a simple and effective solution. No fences are inserted after loads since it is not required on an x86 host system. In order to get a rough estimate of the overhead of fence emulation on a host system with a relaxed memory consistency model (such as POWER), we assumed that the x86 host has a relaxed memory model, and modified COREMU to insert a memory fence after every load and store instruction in the guest application. We use the x86 `mfence` instruction to enforce the required ordering in all our fence emulation implementations.

We also modified COREMU to execute the guest code as transactions using HLE support available on the host system. This simplifies our implementation since we do not have to generate fallback code or handle forward progress issues that might arise from using HTM support instead (Section 3.2). We implement transactional support at both the translation block and the trace level in order to evaluate them. We handle guest applications with conditional synchronization that can lead to a livelock or deadlock when emulated using transactions by monitoring the transaction abort rate using hardware performance counters and re-translating the guest application using memory fences if the abort rate if very high. We do not encounter such behavior with the evaluated real-world applications, and the kernels used to verify correctness, since they do not have such conditional synchronization constructs.



(a) 2 threads



(b) 4 threads

Fig. 9: Execution time of the applications on the virtual machine using transactions (trace level), and memory fences inserted assuming a relaxed host system, normalized to execution time with fences inserted only after a store instruction.

4.1. Performance Comparison

Figure 9(a) compares the performance of emulation using memory fences and transactions. The figure shows the execution times of the applications on the virtual machine, emulated with transactions formed at the trace boundaries, normalized to the execution times when emulated by inserting memory fences only after a store instruction. The figure also shows the execution times of the applications emulated using memory fences assuming that the x86 host has a weak memory consistency model (by inserting a memory fence after every store *and* load instruction), normalized to the same baseline. Therefore, in the *fence - relaxed host* configuration, a memory fence is inserted after every memory operation in the guest application, while in the baseline system a fence is inserted only after every store instruction in the guest application. Figure 9(b) shows the same data for 4-thread applications.

The transactional execution results demonstrate that there is a variation in the behavior of different applications. Transactional emulation is faster than the baseline for 2-thread applications such as RADIOSITY (0.74), RAYTRACE (0.97), WATER (0.77), RADIX (0.81), BLACKSCHOLES (0.9) and SWAPTIONS (0.9). However, the baseline fence emulation is faster for BARNES (1.08), FMM (3.35), OCEAN (1.29), LU (1.05), CHOLESKY (4.06), VOLREND(4.56), DEDUP (1.41), FACESIM(1.30), FLUIDANIMATE (1.11), FREQMINE (1.25), STREAMCLUSTER (1.19) and X264 (1.11). The trends are similar for most

Application	Inst. per TX	(LD + ST) per TX	ST per TX	Abort rate (%)
BARNES	43.36	24.36	8.84	46.67
FMM	202.67	36.38	4.33	89.52
LU	8778.67	3445.00	984.33	99.52
OCEAN	185.81	66.04	0.65	90.41
RADIOSITY	38.02	4.63	4.51	2.77
RAYTRACE	17.45	6.82	0.55	1.67
WATER	69.28	27.66	7.24	11.38
FFT	387.38	128.50	45.88	92.66
RADIX	82.68	13.71	4.94	22.73
CHOLESKY	313.00	119.00	29.33	96.55
VOLREND	69.92	23.13	4.50	85.41
BLACKSCHOLES	22.69	6.89	1.96	3.06
CANNEAL	16.69	6.61	3.27	2.00
DEDUP	45.91	20.35	6.36	53.97
FACESIM	49.65	21.90	6.55	55.50
FLUIDANIMATE	32.92	11.15	1.76	25.99
FREQMINE	46.59	21.04	6.85	37.88
STREAMCLUSTER	18.40	7.73	0.25	10.57
SWAPTIONS	26.87	10.30	2.66	24.71
X264	28.32	10.13	2.46	18.99

Table III: Characteristics of the transactions formed during emulation using transactions. LD stands for number of load instructions, ST stands for number of store instructions, and TX stands for transaction.

applications when run with 4 threads. Transactional emulation and emulation using the baseline fence configuration are comparable for FFT (1.00) with 2 threads; however the baseline is faster in the case of the 4-thread version. In the case of CANNEAL (1.00), the baseline and transactional emulation configurations are comparable for both 2 and 4 threads. These results show that the best technique depends on the characteristics of the emulated application.

The *fence - relaxed* emulation results show that, as expected, the execution times of most applications are much slower when a fence is inserted after every memory operation in the application. Moreover, unlike transactional emulation, the *fence - relaxed* execution times do not vary between the 2 and 4 thread applications; this is also expected since fence emulation overhead depends mainly on the number of memory operations per thread, rather than the number of threads in the application. The results show that transactional emulation is faster than *fence - relaxed* emulation for most of the applications in both the 2 and the 4 thread cases. Therefore, the *fence - relaxed* results suggest that transactional emulation can be more beneficial than fence emulation across a wide range of applications on a host system with a relaxed memory consistency model. We use the *fence - relaxed* results solely to illustrate the potential benefits of transactional emulation on a relaxed host system. Since inserting a fence after every memory operation is not required on an x86 host system, and doing so can artificially inflate the overhead of fence emulation, we do not include these results in the rest of the paper. For the rest of this paper, we refer to the baseline fence emulation configuration as simply fence emulation.

Table III lists the characteristics of the transactions formed in each application. Note that the transactions are formed at the trace boundaries. The table shows the average

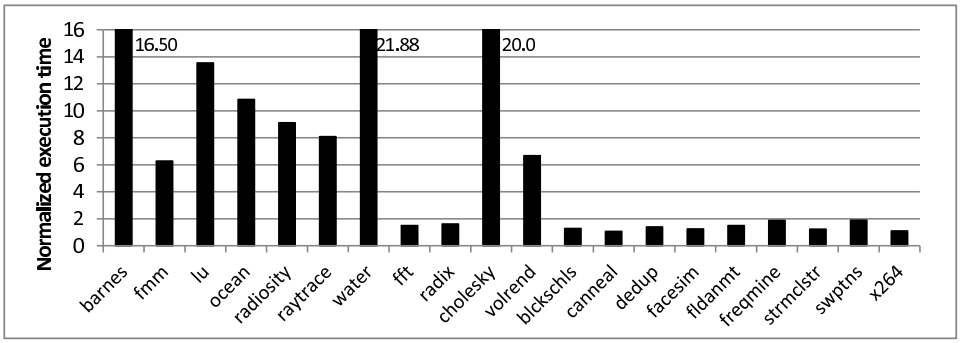


Fig. 10: Execution time of the applications on the virtual machine with transactions formed at the translation block boundaries normalized to the execution time with transactions formed at trace boundaries.

number of guest instructions, guest memory accesses and guest stores per transaction in the evaluated applications. It also shows the abort rate of the transactions for each application when run with 2 threads.

Transactional emulation results in poor performance in BARNES, FMM, OCEAN, CHOLESKY, VOLREND, DEDUP, FACESIM, FLUIDANIMATE and FREQMINE due to the high abort rate. Transactions abort in these applications due to data conflicts. Apart from true data dependency conflicts, false sharing in these applications also results in aborts since Haswell tracks dependencies at the cache block level. Transactional emulation in FFT and LU has a high abort rate, but its performance is comparable to emulation using fences since the overhead of fence emulation is also large due to the high number of stores per transaction in these applications. In the case of STREAMCLUSTER transactional emulation is slower than fence emulation, even with a fairly low abort rate, since the fence overhead is very low given the small number of stores per transaction.

Transactional emulation is faster than using fences in BLACKSCHOLES and SWAPTIONS since the abort rate in these applications is fairly low. Transactional emulation outperforms emulation using fences in RADIOSITY, WATER and RADIX because of two reasons. These programs have very low abort rates leading to a low overhead and, the number of stores per transaction in these applications is also fairly large resulting in a high overhead when using memory fences. Transactional emulation is only marginally faster in RAYTRACE, although it has a low abort rate, since the number of stores per transaction in the program is small thereby resulting in a low overhead when emulating using fences. In the case of CANNEAL, the two approaches are comparable since the execution time of the emulated application is dominated by the initial phase where the main thread reads the input data.

Figure 10 shows the effect of the transaction size on emulation. It shows the execution time of the applications on the virtual machine when emulated with transactions formed at the translation block boundaries normalized to execution time with transactions formed at the trace boundaries. The results are shown for 2-thread applications. The results show that emulation with transactions formed at translation block boundaries is significantly slower with as much as 20x slowdown (WATER). This is because in most of the applications translation blocks are just a few instructions in length, and transactions at the translation block boundaries are not large enough to amortize the overhead of starting and ending a transaction. There is a marked difference between the SPLASH-2 and the PARSEC applications. In the PARSEC applications, the differ-

ence in the number of instructions per trace and per translation block is not as large as in the SPLASH-2 applications. However, transactional emulation at the translation block level is still slower than emulation at the trace level in the PARSEC applications with as much as 1.89x slowdown (FREQMINE and SWAPTIONS). Since emulation with transactions formed at the translation block boundaries results in poor performance, in the rest of this paper we focus only on transactional execution with transactions formed at the trace level.

4.2. Hybrid Emulation Using Fences and Transactions

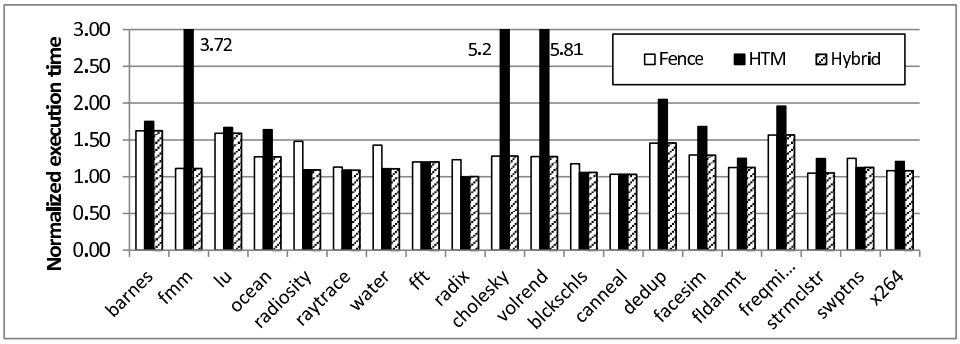
Figure 11 shows the execution time of the applications on the virtual machine using memory fences, transactions, and our hybrid technique, all normalized to the execution time of the applications without any support. Although emulating an application without any support can lead to an incorrect emulation, we choose it as the baseline to illustrate the overhead of each emulation technique. The data is shown for both 2-thread and 4-thread applications. The results show that the hybrid technique chooses the best approach for all the applications. Most of the evaluated applications exhibit bipolar behavior with one technique resulting in much better performance than the other. Therefore, the proposed simple profiling technique is sufficient in order to choose the best policy. The profiling overhead for the hybrid technique is less than 1% and does not result in a slowdown. The average overhead for emulation using fences, transactions, and the hybrid technique, compared to the incorrect baseline emulation, is 27.1%, 60.8%, and 20.8% for 2-thread applications; and 32.3%, 128.4%, and 26.3%, respectively for 4-thread applications. On average, memory consistency model emulation using the proposed hybrid technique is 4.9% faster than emulation using fences and 24.9% faster than emulation using transactions for 2-thread applications; and 4.5% and 44.7%, respectively, for 4-thread applications.

4.3. Overhead of Memory Consistency Model Emulation

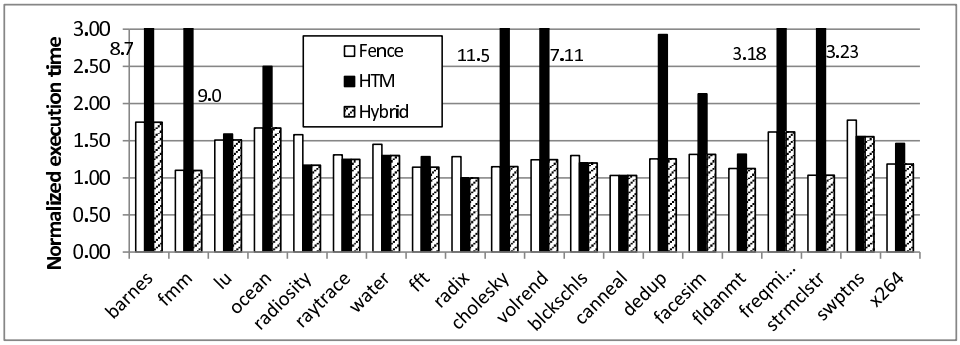
Figure 12 shows the execution time of the applications on the virtual machine, emulated using the hybrid technique, normalized to the native execution time. The normalized time is split to show the contribution of the overhead of memory consistency model emulation to the total overhead of system virtualization. The data is shown for both 2- thread and 4-thread applications. On average, the total virtualization overhead using the hybrid technique is 24.5x for 2-thread applications and 25.8x for 4-thread applications. The results show that in most applications the overhead of memory consistency model emulation is a small, but non-trivial fraction of the total overhead of system virtualization. On average, memory consistency model emulation contributes 11.3% and 13.9% of the total system virtualization overhead for 2-thread and 4-thread applications, respectively. The overhead of memory consistency model emulation can be decreased by selectively applying the emulation technique to only shared variable accesses in the application. However, in order to filter the accesses to private data, the emulator needs access to high level program semantic information. Incorporating program semantic information in emulators, using compiler or binary analysis, is left as future work.

4.4. Discussion

Our evaluation illustrates the validity of memory consistency model emulation using fences and transactions, and highlights the performance tradeoffs between the two approaches. It also shows that the hybrid technique proposed in this work can correctly choose the approach with the lowest overhead. Thus, although our evaluation uses a guest-host pair that differ only in their memory consistency models, our proposed technique and the performance tradeoffs between fence and transaction emulation are



(a) 2 threads



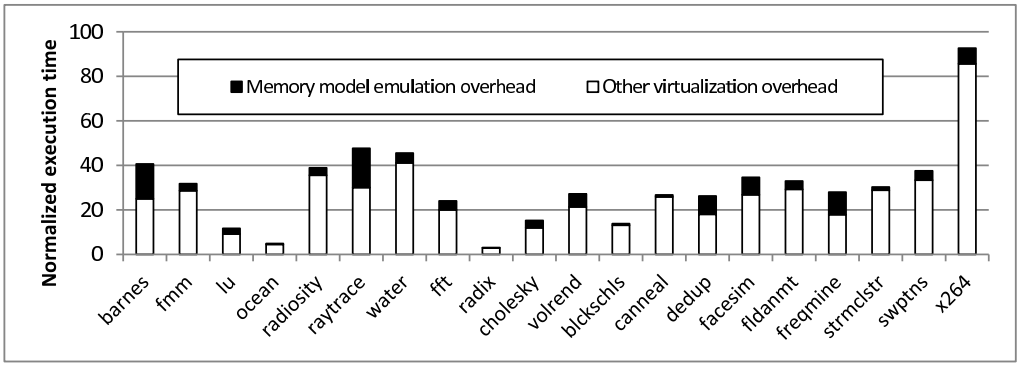
(b) 4 threads

Fig. 11: Execution time of the applications on the virtual machine using transactions (trace level), memory fences, and hybrid techniques normalized to execution time without any support (incorrect emulation).

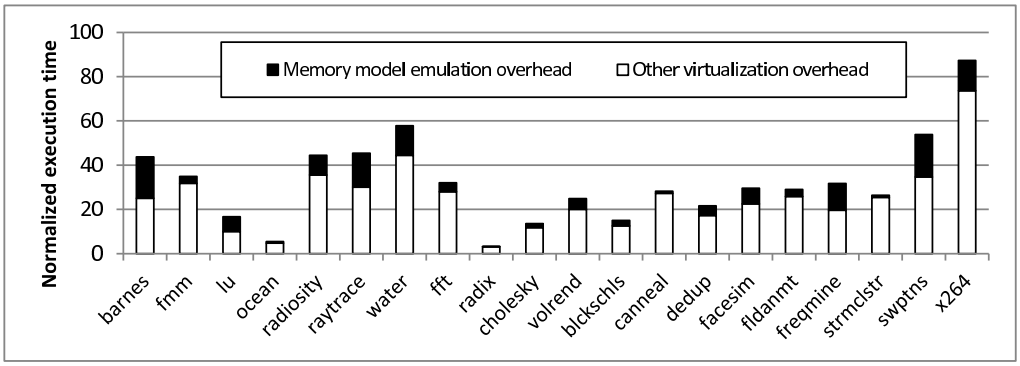
valid on a real cross-ISA system where both the instruction set and the memory consistency models of the guest-host pair differ.

Transactions formed at the trace level can effectively hide the overhead of starting and ending a transaction. Therefore, the overhead of transactional emulation depends mainly on the abort rate. Since the transaction abort rate is an application characteristic, we expect the overhead of emulating an application using transactions to be similar to the results shown in this paper in a real cross-ISA system. The overhead of fence emulation, on the other hand, depends on the number of memory operations, which is an application characteristic, as well as the placement of the fences in the translated code, which depends on the guest and host memory consistency models. Hence, the overhead of emulating an application using fences might vary from the results shown in this paper depending on the host and guest ISA pair. Although the technique with the lowest overhead for an application might change in a different guest-host ISA pair, the proposed hybrid technique would still be able to correctly identify it.

The total overhead of system virtualization is likely to increase in a real cross-ISA system due to the increased instruction translation time. The overhead of memory consistency model emulation in a real cross-ISA system can increase in cases where the hybrid technique employs fence emulation, but it would be similar for applications where transactional emulation is chosen by the hybrid technique. Thus, we expect the contribution of the overhead of memory consistency model emulation to the total



(a) 2 threads



(b) 4 threads

Fig. 12: Execution time of the applications on the virtual machine using the hybrid technique normalized to the native execution time. The normalized time is split to show the contribution of the overhead of memory consistency model emulation to the total virtualization overhead.

overhead of system virtualization to be similar to the results shown in Figure 12 in a real cross-ISA system.

5. RELATED WORK

Previous works have explored system virtualization of multithreaded applications. Sequential system emulators, which emulate multithreaded applications by time-sharing emulated threads on a single physical core on the host system, have been proposed previously [Bellard 2005; Magnusson et al. 2002; Bochs 2014]. In such emulators the memory consistency model of the guest system is inconsequential since only one thread is emulated at a time on the host system. Therefore, sequential emulators can emulate any guest-host memory consistency model pair. However, they suffer in performance since they do not utilize the resources available on current multicore systems. Parallel system emulators, which run multiple emulated threads simultaneously on multiple physical cores on the host system, greatly increase emulation speed [Wang et al. 2011; Ding et al. 2011]. But such emulators only support same-ISA guest-host pairs or support only guest systems that have weaker memory consistency models than the host systems. The techniques proposed in this paper are orthogonal to these works. They

can be applied to existing parallel system emulators to extend them to support a wider range of guest-host pairs.

Techniques for automatic placement of fences in parallel applications running on relaxed memory systems have been explored in previous work. The delay set analysis algorithm is used widely for inferring the placement of memory fences in parallel applications on relaxed memory systems [Shasha and Snir 1988]. Various compiler techniques and automated tools for inserting fences based on the delay set algorithm have been proposed [Burckhardt et al. 2007; Kuperstein et al. 2010; Fang et al. 2003; Duan et al. 2009]. However, such techniques are aimed at helping developers write concurrent programs for relaxed memory consistency models, and rely on static or dynamic program analysis, memory consistency model descriptions or program inputs. The limited availability of program semantic information at runtime, and the high cost of these techniques makes them unsuitable for use in emulators. The memory fence insertion techniques discussed in this paper are simple, fast, and low cost techniques suitable for runtime systems.

The idea of executing memory accesses as coarse-grained, sequentially consistent chunks has been proposed as a solution for enforcing sequential consistency on modern processors without sacrificing performance [Ceze et al. 2007; Galluzzi et al. 2007; Hammond et al. 2004a; Hammond et al. 2004b]. These prior works focus on the problem of enforcing sequential consistency on modern processors while our work focuses on memory consistency model emulation. We do not propose any hardware changes and instead leverage existing hardware on processors.

Using transactional memory has been previously proposed as a solution for thread-safe dynamic binary translation of multi-threaded applications [Chung et al. 2008]. The authors propose using transactional memory to eliminate data races among metadata maintained by dynamic binary translation tools in multithreaded applications. In contrast, our work proposes using transactional execution as a solution for memory consistency model emulation.

6. CONCLUSION

In this paper we focus on memory consistency model emulation in virtual machines where the memory consistency models of the guest and the host systems differ. To the best of our knowledge this is the first work that addresses this issue in system virtualization. We compare the performance impact of two mechanisms for emulating memory consistency models: memory fence insertion and execution transactionalization. We implement the two mechanisms on COREMU, a parallel emulator. Our experimental results show that, on microprocessors with adequate hardware support for transactionalizing instruction sequences, transactional execution is a viable alternative to memory fence insertion for certain workloads. Thus, we propose and evaluate a hybrid approach that dynamically determines whether to emulate the memory consistency model by inserting fence instructions or through transactional execution. We evaluate the proposed hybrid approach on real-world parallel applications from the PARSEC and the SPLASH-2 benchmark suites. Our evaluation demonstrates that the hybrid approach is able to outperform the fence insertion mechanism by 4.9% and the transactional execution approach by 24.9% for 2-thread applications; and outperform them by 4.5% and 44.7%, respectively for 4-threaded execution.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their constructive feedback which greatly helped in improving the quality of this paper. We would also like to thank Pen-Chung Yew, Anup Holey, Vineeth Mekkat, and Jieming Yin for their feedback at various stages of this work.

REFERENCES

Sarita V. Adve and Kourosh Gharachorloo. 1996. Shared Memory Consistency Models: A Tutorial. *Computer* 29, 12 (Dec. 1996), 66–76.

Fabrice Bellard. 2005. QEMU, a Fast and Portable Dynamic Translator. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference (ATEC '05)*. USENIX Association, Berkeley, CA, USA, 41–41.

Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. 2008. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*.

Colin Blundell, E Christopher Lewis, and Milo Martin. 2005. Deconstructing transactional semantics: The subtleties of atomicity. *Annual Workshop on Duplicating, Deconstructing, and Debunking (WDDD)* (2005), 48–55.

Bochs. 2014. (2014). http://bochs.sourceforge.net.

Daniel Bovet and Marco Cesati. 2005. *Understanding The Linux Kernel*. Oreilly & Associates Inc.

Sebastian Burckhardt, Rajeev Alur, and Milo M. K. Martin. 2007. CheckFence: Checking Consistency of Concurrent Data Types on Relaxed Memory Models. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '07)*. ACM, New York, NY, USA, 12–21.

Harold W. Cain, Maged M. Michael, Brad Frey, Cathy May, Derek Williams, and Hung Le. 2013. Robust Architectural Support for Transactional Memory in the Power Architecture. In *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA '13)*. ACM, New York, NY, USA, 225–236.

Luis Ceze, James Tuck, Pablo Montesinos, and Josep Torrellas. 2007. BulkSC: Bulk Enforcement of Sequential Consistency. In *Proceedings of the 34th Annual International Symposium on Computer Architecture (ISCA '07)*. ACM, New York, NY, USA, 278–289.

Jaewoong Chung, M. Dalton, H. Kannan, and C. Kozyrakis. 2008. Thread-safe dynamic binary translation using transactional memory. In *High Performance Computer Architecture, 2008. HPCA 2008. IEEE 14th International Symposium on*. 279–289.

Dave Dice and Nir Shavit. 2010. TLRW: Return of the Read-write Lock. In *Proceedings of the Twenty-second Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '10)*. ACM, New York, NY, USA, 284–293.

Edsger Wybe Dijkstra. 1965. Cooperating Sequential Processes, Technical Report EWD-123. (1965).

Jiun-Hung Ding, Po-Chun Chang, Wei-Chung Hsu, and Yeh-Ching Chung. 2011. PQEMU: A Parallel System Emulator Based on QEMU. In *Parallel and Distributed Systems (ICPADS), 2011 IEEE 17th International Conference on*. 276–283.

Yuelu Duan, Xiaobing Feng, Lei Wang, Chao Zhang, and Pen-Chung Yew. 2009. Detecting and Eliminating Potential Violations of Sequential Consistency for Concurrent C/C++ Programs. In *Proceedings of the 7th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO '09)*. IEEE Computer Society, Washington, DC, USA, 25–34.

Xing Fang, Jaejin Lee, and Samuel P. Midkiff. 2003. Automatic Fence Insertion for Shared Memory Multiprocessing. In *Proceedings of the 17th Annual International Conference on Supercomputing (ICS '03)*. ACM, New York, NY, USA, 285–294.

Marco Galluzzi, Enrique Vallejo, Adrián Cristal, Fernando Vallejo, Ramón Beivide, Per Stenström, James E. Smith, and Mateo Valero. 2007. Implicit Transactional Memory in Kilo-instruction Multiprocessors. In *Proceedings of the 12th Asia-Pacific Conference on Advances in Computer Systems Architecture (ACSAC'07)*. Springer-Verlag, Berlin, Heidelberg, 339–353.

Lance Hammond, Brian D. Carlstrom, Vicky Wong, Ben Hertzberg, Mike Chen, Christos Kozyrakis, and Kunle Olukotun. 2004a. Programming with Transactional Coherence and Consistency (TCC). In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XI)*. ACM, New York, NY, USA, 1–13.

Lance Hammond, Vicky Wong, Mike Chen, Brian D. Carlstrom, John D. Davis, Ben Hertzberg, Manohar K. Prabhu, Honggo Wijaya, Christos Kozyrakis, and Kunle Olukotun. 2004b. Transactional Memory Coherence and Consistency. In *Proceedings of the 31st Annual International Symposium on Computer Architecture (ISCA '04)*. IEEE Computer Society, Washington, DC, USA, 102–.

Intel Corporation. 2013. Intel Architecture Instruction Set Extensions Programming Reference. (2013). http://intel.com.

Christian Jacobi, Timothy Slegel, and Dan Greiner. 2012. Transactional Memory Architecture and Implementation for IBM System Z. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-45)*. IEEE Computer Society, Washington, DC, USA, 25–36.

Michael Kuperstein, Martin Vechev, and Eran Yahav. 2010. Automatic Inference of Memory Fences. In *Proceedings of the 2010 Conference on Formal Methods in Computer-Aided Design (FMCAD '10)*. FMCAD Inc, Austin, TX, 111–120.

L. Lamport. 1979. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *Computers, IEEE Transactions on* C-28, 9 (1979), 690–691.

Changhui Lin, Vijay Nagarajan, and Rajiv Gupta. 2010. Efficient Sequential Consistency Using Conditional Fences. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques (PACT '10)*. ACM, New York, NY, USA, 295–306.

P.S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner. 2002. Simics: A full system simulation platform. *Computer* 35, 2 (2002), 50–58.

Luc Maranget, Susmit Sarkar, and Peter Sewell. 2012. A Tutorial Introduction to the ARM and POWER Relaxed Memory Models. (2012).

Scott Owens, Susmit Sarkar, and Peter Sewell. 2009. A better x86 memory model: x86-TSO. In *In TPHOLs09: Conference on Theorem Proving in Higher Order Logics, volume 5674 of LNCS*. Springer, 391–407.

Gary L. Peterson. 1981. Myths About the Mutual Exclusion Problem. *Inf. Process. Lett.* 12, 3 (1981), 115–116.

Ravi Rajwar and James R. Goodman. 2001. Speculative Lock Elision: Enabling Highly Concurrent Multithreaded Execution. In *Proceedings of the 34th Annual ACM/IEEE International Symposium on Microarchitecture (MICRO 34)*. IEEE Computer Society, Washington, DC, USA, 294–305.

Carl G Ritson and Frederick RM Barnes. 2013. An Evaluation of Intel's Restricted Transactional Memory for CPAs. *Communicating Process Architectures 2013* (2013), 271–291.

Dennis Shasha and Marc Snir. 1988. Efficient and Correct Execution of Parallel Programs That Share Memory. *ACM Trans. Program. Lang. Syst.* 10, 2 (April 1988), 282–312.

Jim Smith and Ravi Nair. 2005. *Virtual Machines: Versatile Platforms for Systems and Processes (The Morgan Kaufmann Series in Computer Architecture and Design)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

Christoph von Praun, Harold W. Cain, Jong-Deok Choi, and Kyung Dong Ryu. 2006. Conditional Memory Ordering. In *Proceedings of the 33rd Annual International Symposium on Computer Architecture (ISCA '06)*. IEEE Computer Society, Washington, DC, USA, 41–52.

Zhaoguo Wang, Ran Liu, Yufei Chen, Xi Wu, Haibo Chen, Weihua Zhang, and Binyu Zang. 2011. COREMU: A Scalable and Portable Parallel Full-system Emulator. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming (PPoPP '11)*. ACM, New York, NY, USA, 213–222.

S.C. Woo, M. Ohara, E. Torrie, J.P. Singh, and A. Gupta. 1995. The SPLASH-2 programs: characterization and methodological considerations. In *Computer Architecture, 1995. Proceedings., 22nd Annual International Symposium on*. 24 –36.